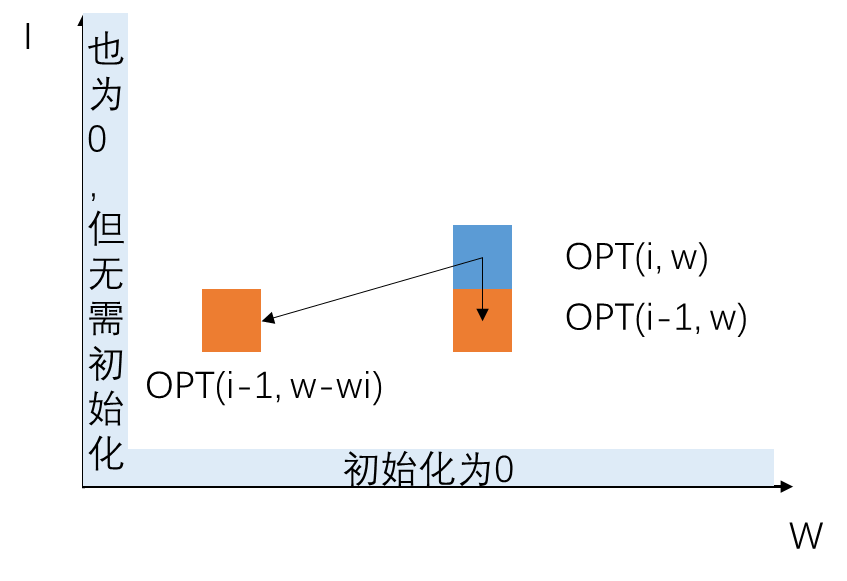
1. **背包问题变种：**

0-1背包问题的关系依赖图：



对于背包问题，绘制关系依赖图能够有效帮助我们分析当前项与哪些项有关系。

**注意：如果循环从w为0开始，则无需初始化第0列，如果从1开始，则需要初始化。**

同时，经典的0-1背包要求背包容量和物品重量都是正整数，这样能作为数组下标进行遍历。否则，如果物品重量有小数，则需要放大转换为整数求解。（100位就需要2100次遍历？）

**空间复杂度优化为θ(w)**：如果只需要求解最大的v值，而不需要输出选择的物品，则我们可以将n\*w的map变成w的数组，每次循环时，从w到0逆序循环，这样保证每次依赖的项都是上一次循环的值。最终输出OPT(n ,w)即可。

**变种1：完全背包问题，将n个物品转换为n种物品，每种物品可以选择任意多个**

**解法1：**转换为0-1背包，因为每个物品有自己的重量，所以最多选择W/wi件，将其展开即可。主要问题是种类到数量的膨胀，使得复杂度变高。

**解法2：**

OPT(i, w)定义不变，仍是前i个物品，重量不超过w时的最优解，但此时由于每个物品可以选多次，则转换方程需要改变：

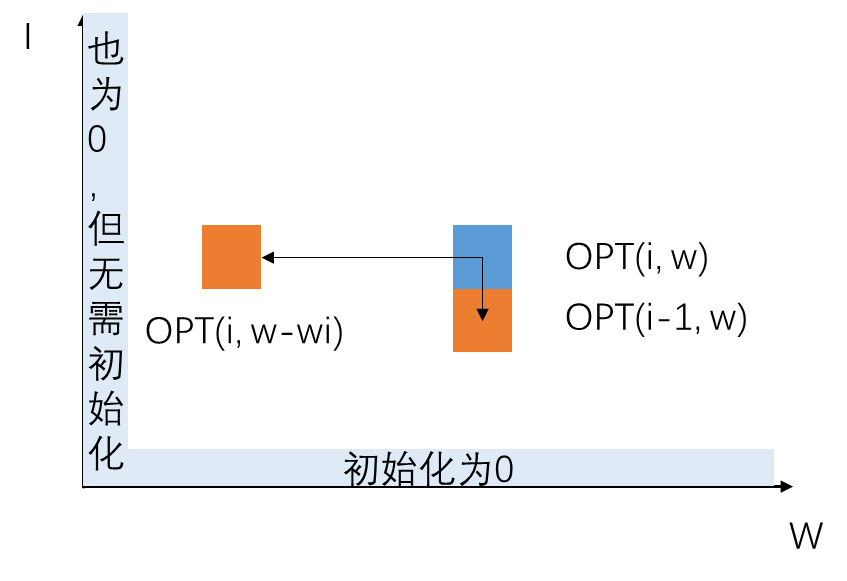
OPT(i, w)=

0, i=0

OPT(i-1, w), w<wi

Max{ OPT(i-1, w), OPT(i, w-wi)+vi}, else

即此时即使选择了物品i，仍然可以继续选择，所以不是OPT(i-1, w-wi)+vi，而是OPT(i, w-wi)+vi。依赖关系图：



时空复杂度均为θ(n\*W)。同样，空间复杂度可优化为θ(w)，此时与0-1背包不同的是，循环需要从0开始到w结束，因为当前依赖的值是本次循环的值，而不是上次循环的值。

**变种2：多重背包问题，每种物品有数量限制**

**解法1：**同样拆分成0-1背包问题

可以优化，比如一件物品的数量是1024件，则可以优化为1，2，4，8，…，512件。更一般地，如果有12件，则可以优化为1，2，4，5件。分割的准则为不超过件数一半的部分按照2的幂划分，最终将剩余部分划分为一个整体，如12一半为6，则1，2，4按照幂划分，8>6，则将12-1-2-4划分为最后一件（这里也可以将5换为8，表示能力更强，且不复杂，找到大于该数一半的第一个幂停止即可）。这样，每件新物品的价值和重量为对应数量的原始物品价值之和，并且每件物品最多选1次就能凑出任意数量的原物品。

该方法类似进制表示，最少用多少位表示一个数。

**解法2：枚举取第k件物品的数量**

显然，对于一个物品i，所有情况就是取0件，取1件，…， 取n[i]件。当然，n[i]\*wi要小于背包的容量。

所以类似0-1背包问题，转移方程为：OPT(i ,w) = max{f[i-1, w-k\*wi]+k\*vi}，其中k在[0,n[i]]之间遍历。

同样：按照这样的思路，可以看出完全背包也可以通过这样的遍历求解，而0-1背包则是该问题的特殊情况，即n[i]=1。所以记住这个方程，就能计算一部分类似的背包问题了。

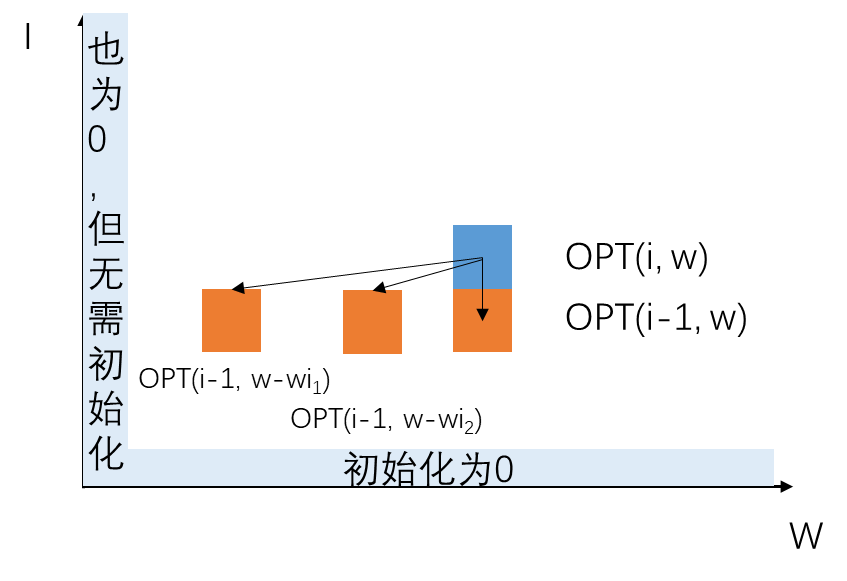
以上两种问题的复杂度均为θ(n\* n[i])，第1种优化后为θ(n\*Σlogn[i])

**变种3：分组背包问题，每组组内物品互相冲突，只能选择一件**

显然，这里我们要简单修改一下OPT(i, w)的定义：前i组中，重量不超过w的最优解。所以对于第i组，要么选择其中一件，要么一件都不选，状态转移方程为：

OPT(i, w)= max { OPT(i-1, w), OPT(i-1, w-wj)+vj 其中j在组内每个物品中遍历}

类似0-1背包，该问题也可以倒着循环，将空间复杂度优化为θ(w)，只不过在求最大值时，依赖多个不同的项。



复杂度为θ(组数\*背包容量\*每组物品个数的平均值)

**变种4：背包恰好被装满时的价值最大值（要考虑是否一定有解）**

首先引出一个推论，即我们之前求的背包问题的解OPT(i, w)都是重量不超过w时解最优解，那么能不能通过不超过w的最优解求出等于w时的最优解呢？答案是否定的。反之，如果我们知道每个等于w时的最优解，能否求出小于等于w时的最优解？可以，遍历找到最大值即可。

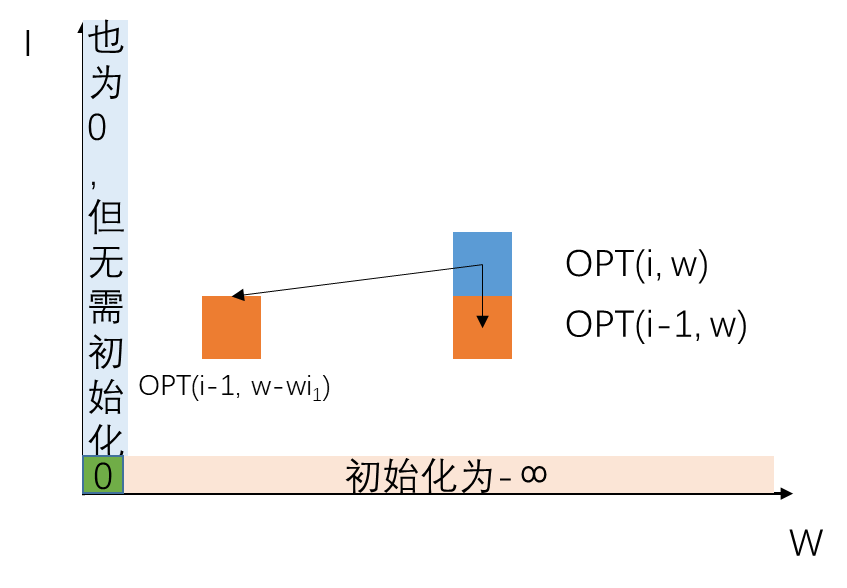
所以按照之前的定义，如果OPT(i, w)和OPT(i, w-1)不等，则OPT(i, w)就是正好装满背包的最优解，但也可能OPT(i, w)与OPT(i, w-1)相等，此时背包一定没有被装满。所以单纯比较二者无法得出正确的结果。

因此我们要修改OPT(i, w)的定义，使其为重量等于w时的最优解。

接着我们思考递推方程：选择第i件物品时，价值为vi+OPT(i-1, w-wi)，不选时价值为OPT(i-1, w)，与0-1背包问题的方程完全一样。实际上，方程形式也的确相同，那么为什么不同的问题可以用同样的动归算法来解决呢？答案是，两个动归算法是否相同不仅仅取决于递推方程，还要看初始化的情况，对于本题，OPT(0,0)=0是正确的，因为什么都不选，背包里的重量自然为0，价值为0。但对于OPT(0, w)，w>0而言，我们要考虑该解是否合理。显然，什么都不选，总重量一定不会等于w，所以这个解是不存在的。我们可以赋值为-1。

与之前相比，OPT(0, w)表示不超过w的物品价值，则可以为0，而等于时解就不合理了，这是两个问题的本质区别。

所以我们进行这样的初始化：对OPT(0,0)初始化为0，OPT(i, 0)初始化为0，OPT(0, w)初始化为负无穷，这样进行递推就能得到最终的解了，如果最终解为负无穷，则说明无法装满背包。



对于负无穷还有一种实现，即用-1来实现，只要递推过程遇到-1，则说明当前解不存在。

上述算法时空复杂度均为θ(n\*W)

1. 背包问题的优化：

使用滚动数组来计算依赖。因为大部分当前次循环的依赖都是上次循环，所以可以通过两个数组互相覆盖来实现存储。但显然这样的存储无法输出最优解的具体内容，并且还能继续优化，即只用一个数组，如上所说，只需要注意循环顺序即可。

此时OPT(w)=max { OPT(w), OPT(w-wi)+vi

不包含第1维。

对于完全背包问题，如果需要输出最优解，我们也不必存储全部的map，而只需要用另外一个数组a，初始化为0，大小为W，当更新OPT(i, w)时，如果采用了OPT(i, w-wi)+vi，则将a[w]赋值为i。当求完最优解后，从后往前遍历arr数组，如果arr[w]=0，则继续遍历arr[w-1]，如果arr[w-1]=i，则输出i，说明选中了第i个物品，继续遍历arr[w-1-wi]。

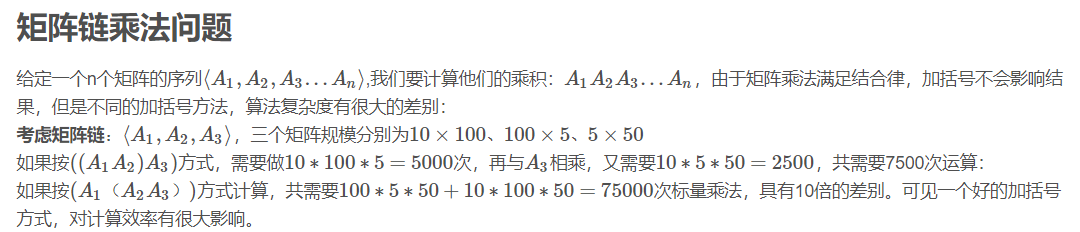
0-1背包不能用类似的方法压缩。

以上不是很懂。

PS. 完全背包问题或分组背包问题如果计算价值最大，则有一个简单的优化。对于完全背包问题，如果存在一个物品A比B的价值更大，重量更小，则可以直接将物品B去掉。同理，对于分组背包，如果同组中两个物品存在上述关系，也可以去掉。

1. **矩阵乘法加括号问题**

问题给定一系列矩阵，要求找到将其相乘时所进行乘法次数最少的加括号方案：



算法：初始化OPT(i, i)=0，因为只有1个矩阵乘时不需要计算。

之后同样也是先计算小的区间，设区间长度为l，起止点为i和j，OPT(i, j)表示矩阵i到j相乘所需的最少乘法次数，则：

for l 2: n //l为对区间长度枚举

for i = 1: n-l+1 //区间左点

j = i+l-1 //区间右点

OPT(i, j)=min{ OPT(i, k) + OPT(k+1, j) + p(i-1)\*p(k)\*p(j)}

其中k=i : j-1，p(i-1)表示第i个矩阵的行数，p(i)表示第i个矩阵的列数，也是第i+1个矩阵的行数

如果需要输出最优的划分方案，可以定义一个矩阵s[i][j]=k，记录s[i][j]对应的最优划分，只要递归查找s[i][k]和s[k+1][j]即可。

https://blog.csdn.net/luoshixian099/article/details/46344175